ISSN 1561-5359. Штучний інтелект, 2018, № 2

УДК 004.415.53+004.416.2

А.В. Колчин, С.В. Потиенко
Институт кибернетики имени В.М. Глушкова НАН Украины, Украина
пр. Академика Глушкова, 40, г. Киев, 03680

ИНТЕРАКТИВНЫЙ МЕТОД АВТОМАТИЗИРОВАННОГО СОЗДАНИЯ ТЕСТОВОГО НАБОРА ДЛЯ ФОРМАЛЬНЫХ МОДЕЛЕЙ ПРОГРАММНЫХ СИСТЕМ

A. Kolchin, S. Potiyenko
V.M. Hlushkov Institute of cybernetics NAS of Ukraine, Ukraine
40, Academician Hlushkov av., Kyiv, Ukraine, 03680 MSP

INTERACTIVE METHOD FOR AUTOMATED TEST SUIT DEVELOPMENT FOR FORMAL MODELS OF SOFTWARE SYSTEMS

Предложен интерактивный метод для упрощения отладки и повышения производительности построения тестовых сценариев для обеспечения покрытия требований высокого уровня. Метод реализует полуавтоматическую генерацию тестов, используя задаваемую пользователем дополнительную информацию об искомых поведениях проектируемой системы. В основу положен эффективный алгоритм сокращения обхода поведения модели, способный оперативно оценивать перспективы достижения непокрытых элементов и приостанавливать рассмотрение ветвей поведения, которые гарантированно не приведут к увеличению покрытия.

Ключевые слова: тестирование, покрытие, редукция пространства поиска

Interactive method for simplification and efficiency improvement of test scenario development to satisfy high-level requirements coverage is proposed. The method implements semiautomatic generation of tests on a basis of points-of-interest of desired behavior of a system under development. It is based on efficient algorithm for state space reducing, which makes on-the-fly assessment of the prospects of achieving elements yet uncovered. The algorithm will suspend consideration of the behavior branches that will provably not be able contribute to coverage.

Key words: testing, coverage, state-space search reduction

Введение

Рассмотрим тестирование программных систем как способ проверки конкретной реализации на соответствие требованиям. Так как при тестировании невозможно проверить поведение программы во всех ситуациях, то во время создания тестов прибегают к определению критериев покрытия и ограничиваются требованием проверки классов тестовых сценариев, удовлетворяющих таким критериям. Часто полнота тестового покрытия оценивается по числу выполненных операторов кода, а также ветвей, путей, достижению граничных значений функций, выполнению условий в выражениях, и т.п. [1, 2]. Подобные критерии удобно использовать для автоматической генерации тестов, которые могут обеспечить структурное покрытие кода разрабатываемого продукта, но не требований к нему. Например, из того, что тестовый набор покрывает все условия и операторы кода не следует, что будет проверена правильность реакции на входящие сигналы. Трудоемкость создания тестов по функциональным спецификациям вручную (или с применением симуляторов) слишком велика для систем со сложной моделью поведения, а качество таких тестов не приемлемо для систем, в которых надежность критична. Обостряется необходимость автоматизации создания тестовых сценариев на основе формальных моделей.

Обычно исходные требования, формулируемые заказчиком, и их детальные спецификации, создаваемые разработчиками, заданы на разных уровнях детализации. Для того чтобы автоматизировать проверку соответствия спецификаций разработчиков требованиям заказчика, необходимо построить формальные модели

спецификаций, согласованные с требованиями заказчика и их интерпретацией разработчиком. Как правило, последние представляют собой сценарии типичных режимов поведения (use-cases) разрабатываемой системы. Такие сценарии можно переформулировать в виде последовательностей событий модели, успешное выполнение которых при определенных условиях будет означать покрытие соответствующего требования [3, 4]. Подобные цепочки событий, соотнесенные с требованием, называют целью теста (test purpose).

Проблемы автоматизации построения тестовых сценариев

Разработка тестового набора, обеспечивающего покрытие требований к продукту, достаточно сложна и вполне сопоставима с трудоемкостью создания кода. Для снижения трудоемкости создания тестов активно применяются методы и системы автоматической генерации тестовых сценариев на основе формальных моделей разрабатываемых систем (model based testing) [2–5]. Большинство из них применяют некоторый структурный критерий покрытия для поиска соответствующего поведения модели. Хотя получаемые в результате тесты обычно слабо связаны со специфическими особенностями кода тестируемой системы, тем не менее, они содержат представительный набор сценариев ее поведения, что позволяет значительно сократить трудозатраты тестирования сложных систем. Обзоры таких работ можно найти в [1–5].

Однако можно встретить отчеты (например, [6, 7]), свидетельствующие о том, что обеспечение структурного покрытия как цель для автоматической генерации тестов может быть ошибочной стратегией: такие метрики должны использоваться для измерения тщательности покрытия построенными тестами, и не обязательно приводят к хорошим результатам, когда используются в роли спецификации для их генерации. Например, в работе [8], авторы добились покрытия MC/DC (Modified Condition/Decision, включен в сертификацию DO-178B,C безопасности оборудования авиационных систем, а также является частью требований в стандарте автомобильной безопасности ISO 26262 Road Vehicles Functional Safety) для модели системы управления полетами и проверили полученные тесты на реализациях, в которых были допущены ошибки. Авторы отмечают, что автоматически сгенерированные тесты обнаруживали относительно мало ошибок, и, в общем, продемонстрировали эффективность ниже, чем у тестов, произведенных случайным образом.

Еще один недостаток автоматического подхода заключается в том, что генерируемый тестовый сценарий не обязательно является хорошим испытанием проверяемого свойства: сгенерированная тестовая последовательность является артефактом стратегии поиска, поэтому она может заканчиваться в середине некоторого интересного поведения, может включать в себя много избыточности. Причинно-следственные связи часто запутанны и сложны, и, более того, путь может даже не содержать состояние, в котором предпосылка требуемого свойства становится истинной. Автоматические тесты нуждаются в ручном сопровождении для определения условий успешного прохождения, а также для последующего обновления в связи с изменениями в коде продукта [9]. В работах [10, 11] подчеркивается, что плохо составленные тесты имеют негативный эффект на их сопровождение.

Проблемы соответствия требованиям и отладки

При проектировании тестовых сценариев актуальна задача обеспечения семантического соответствия между тестами и критериями покрытия исходных функциональных требований к программному продукту. Системы, основанные на методе model checking, предполагают, что искомый сценарий можно задать в виде некоторой формулы темпоральной логики: если заданная формула станет ложной на некотором пути, то такой путь будет выдан верификационной системой в качестве контр-примера и может рассматриваться как один из сценариев покрытия исходного

требования. Однако на практике часто оказывается, что однозначного соответствия между формулой над атрибутами модели и желаемой цепочкой событий (обусловленной исходными требованиями) нет; более того, формулы становятся громоздкими и сложными для построения. Существуют подходы, которые помимо формальной модели на вход принимают сценарий тестирования, заданный пользователем в виде последовательности сообщений, которыми обмениваются компоненты модели, например, в виде MSC [4].

Также актуальны проблемы отладки как самого тестового сценария, так и анализа результатов выполнения тестов на реализации системы: для устранения дефекта необходимо иметь эффективные средства его локализации.

Проблемы производительности

Автоматизация тестирования – популярная тема современных исследований. Активно разрабатываются различные эволюционные, стохастические и эвристические методы для ускорения достижения покрытия, а также методы, основанные на символьных вычислениях [2], так, развитие SMT алгоритмов позволило существенно повысить производительность так называемого Bounded Model Checking подхода [12]. Однако проблема комбинаторного взрыва остается основным препятствием на пути интеграции формальных методов в разработку промышленных программных систем. Существующие алгоритмы часто углубляются в перебор вариантов, не приводящих к обнаружению новых элементов искомого покрытия. Актуальна задача разработки алгоритмов для эффективного анализа пространства поведения модели, в частности, проблемой становятся циклы, приводящие к бесконечным путям.

Описание метода

Основные цели предлогаемого метода:

- повысить уровень семантического соответствия между генерируемыми тестами и требованиями к разрабатываемой системе;
- упростить и усовершенствовать ручное управление, средства отладки, снабдить информацией о недостающем покрытии и о путях его достижения;
- повысить производительность алгоритмов поиска поведений модели, удовлетворяющих заданным ограничениям.

В практике проектирования качественных программных продуктов разработчикам необходимо согласовать с заказчиком семантику требований и формализовать процедуру их тестового покрытия. Опыт показывает, что наиболее удобной и понятной для обеих сторон запись такого согласования выражается в терминах событий исходной модели. Так, для каждого требования формулировались специальные конструктивные критерии [13], которые, с одной стороны, подтверждали семантическое соответствие требованию, с другой – служили дополнительным входом для автоматической генерации тестов.

Описываемый метод развивает работы [14, 15], в которых был предложен метод направленного поиска. Требования к искомому поведению ассоциировались с некоторым регулярным выражением над алфавитом, семантически соответствующим событиям модели. К недостаткам можно отнести высокую сложность создания таких выражений, ведь они должны учитывать «дистанцию» между описываемыми событиями (таким образом накладывалось ограничение на поиск, ввиду проблемы комбинаторного взрыва). Также был затруднен анализ результатов для случая, когда искомое поведение оказывалось недостижимым ввиду чрезмерно строгих ограничений, накладываемых на поиск.

В предлагаемом методе цель теста (запрос или условия для выбора) также задается пользователем и определяет свойство искомого пути в виде множества точек потока управления, опционально упорядоченное и ограниченное количеством их посещений, а также расширенное условиями над переменными модели.

Например, условие x>0 на выбранной точке будет означать «рассматривать только пути, в которых значения переменной x в данной точке больше 0»; чтобы исключить из рассмотрения пути, проходящие через некоторую точку, достаточно установить количество ее прохождений в 0.

Идея упрощения поиска путей состоит в том, чтобы оперативно предоставить разработчику тестового набора информацию о множестве всех допустимых поведений модели. Наивное решение – генерация всех путей – это, очевидно, неосуществимая задача, и даже если множество конечно, оно может быть необозримым из-за огромных размеров. Описанный метод предлагает компромисс: он исходит из предположения, что основные интересующие события представлены различными конструктивными элементами дизайна модели (т.е. они находятся в разных точках потока управления), и поэтому проекция искомого пути на структуру потока управления (или блок-схему) будет информативной для разработчика. Так, в начале работы (до внесения каких-либо ограничений на поиск) отображается информация обо всех достижимых траекториях путем визуализации их проекции на поток управления. Далее, пользователь шаг за шагом интерактивно задает дополнительные ограничения, при этом реакцией служит обновление информации о допустимых поведениях. В итоге получается, что пользователь задал некоторую последовательность событий, ассоциированную с требованием к реализуемой системе, и получил в ответ проекцию всего множества трасс (последовательности переходов) модели, каждая из которых, выходя из начального состояния, пройдет через все точки в (опционально) указанной последовательности в соответствии с заданными ограничениями. Это дает пользователю возможность визуального контроля всех возможных альтернатив поведения одновременно (путем выделения соответствующих элементов графического потока на панели визуализации). Таким образом, метод позволяет разработчику найти нужный путь, итеративно шаг за шагом задавая его ключевые точки, тогда как все возможные альтернативы промежутков между ними будут генерироваться автоматически, а проекция полных допустимых путей обновляться «на лету». После визуального контроля можно записать на диск множество трасс, каждая из которых удовлетворяет требованию и которые в совокупности обеспечивают покрытие всех выделенных ветвей потока управления.

Для отладочных целей при записи трассы сохраняется информация обо всех причинно-следственных связях в ней, т.е. места определения значений переменных для каждого использования в условиях или правых частях присваиваний. Эти данные упощают отладку модели и, в последствии, локализацию дефектов, обнаруженных по результатам выполенния тестов.

Также накапливается информация о достигнутом покрытии для подсказки пользователю и приоритезации поиска направлений, содержащих больше непокрытых элементов.

Описание алгоритма обхода поведения модели

В основу метода положены алгоритмы обхода пространства поведения модели [16, 17], эффективно генерирующие проекцию всех выполнимых путей. Путь (последовательность переходов модели) удовлетворяет запросу, если он включает все заданные контрольные точки в соответствии с условиями.

В качестве примера существующих подходов рассмотрим алгоритм генерации тестовых наборов применительно для автоматных моделей с семантикой конечных транзиционных систем. На рис.1 представлен алгоритм [18], учитывающий частичное покрытие рассматриваемой трассой элементов, требуемых заданным критерием

PASS:= $\emptyset$; WAIT:= $\{(s_0, A_0, C_0, \epsilon)\}$; SUITE:= $\emptyset$; COV := C_0
while WAIT$\neq \emptyset$ **do**
  select (s, A, C, ω) from WAIT; add (s, A, C, ω) to PASS
  for all $(s', A', C', \omega.t) : (s, A, C, \omega) \overset{t}{\Rightarrow}_c (s', A', C', \omega.t)$ **do**
    if $C' \not\subseteq$ COV **then**
      add $(\omega.t, C)$ to SUITE; COV := COV $\cup\, C'$
    if $\neg\exists(s_i, A_i, C_i, \omega_i) : (s_i, A_i, C_i, \omega_i) \in$ PASS $\cup$ WAIT $\wedge\ s_i = s' \wedge A' \subseteq A_i$ **then**
      add $(s', A', C', \omega.t)$ to WAIT
  od
od
return SUITE

Рис. 1. Алгоритм покрытия с учетом частичных элементов [18]

Алгоритм заканчивает работу, когда множество нерассмотренных состояний WAIT становится пустым. К этому моменту все достижимые состояния из начального состояния s_0 рассмотрены, множество Cov содержит все достижимые элементы покрытия и SUITE содержит множество пар вида (w_i, C_i), где w_i – трасса, заканчивающаяся покрытием элемента C_i, и $\bigcup_i C_i$ = Cov. Алгоритм обеспечивает полное покрытие достижимых элементов, т.к. для каждого частичного элемента a_i расширенное состояние (s, A, C, w) такое, что $a_i \in A$, будет рассмотрено.

Хорошо известной проблемой подобных алгоритмов является время, затраченное на обход пространства поведения и память для хранения пройденных состояний. Для поставленной задачи пространство поиска, порожденное алгоритмом [18], будет иметь размер, определяемый произведением количества состояний модели |S|, количества возможных подмножеств множества точек покрытия $2^{|R|}$ и количества точек потока управления |C|. Очевидная оптимизация – рассматривать включение $A' \subseteq A_i$ вместо равенства $A'=A_i$ [18]. Однако производительность остается неприемлемой, и, как следствие, алгоритм не пригоден для предлагаемого подхода.

Ключевая идея усовершенствования поиска в прогнозировании перспектив рассматриваемого состояния заключается в следующем: текущее состояние поиска будет рассматриваться как неперспективное (и, как следствие, текущий путь будет остановлен), если никакое его продолжение не сможет достичь непокрытый эле-мент. С другой стороны, если состояние, покрывающее искомый элемент достижимо, оно будет рассмотрено алгоритмом, таким образом сохраняя свойство полноты результатов поиска. Необходимо отметить, что такая остановка развертывания состояний в результате дает большой прирост эффективности поиска, так как количество путей, выходящих из состояния, может экспоненциально возрастать с увеличением встречающихся условий. Детальное описание алгоритма поиска приведено в работе [16].

Рассмотрим пример, демонстрирующий экспоненциальное сокращение анализируемого пространства поведения модели. Пусть для некоторой абстрактной программы (см. рис. 2) множество точек, требуемых для посещения трассой, R={w,a_1,…,a_n}. Асимптотическая оценка количества различных состояний $O(|S|\cdot|C|\cdot 2^{|R|})=O(n^2\cdot 2^n)$, однако алгоритм [16] закончит работу за $O(n^2)$ шагов: действительно, пути {init,a_1,…,a_{n-2},b_{n-1},c_{n-1}}, {init,a_1,…,a_{n-3},b_{n-2},c_{n-2}}, …, {init,b_1,c_1} не будут продолжены, так как станет известно, что элемент ‘w’ не будет достигнут на их продолжении.

```
statement init;
if(cond0){
    statement W;
}else{
   if(cond1)
      statement a1;
   else
      statement b1;
   statement c1;
   …
   if(condn)
      statement an;
   else
      statement bn;
   statement cn;
}
```

Рис.2. Пример программы

Аналогичный подход к редукции пространства поиска предложен в работе [19] применительно для символьного выполнения. Его авторы отмечают, что состояния, отличающиеся только значениями переменных, которые в последствие (на развертываниях всех выходящих путей) не читаются, будут иметь одинаковые последующие выполнения, следовательно, второе выполнение будет избыточным. Стоит отметить, однако, что алгоритм [19] не способен корректно выявлять все читаемые переменные для случая программ с бесконечными циклами, как следствие, авторы указывают на необходимость их ограниченного развертывания. Для устранения последнего недостатка в алгоритме [16] предусмотрена процедура *уточнения* состояния и его последующего повторного анализа.

Заключение

Обычно автоматически сгенерированные тесты позволяют проверить только неявные требования, например, отсутствие зависаний, аварийных завершений или неожиданных исключительных ситуаций во время выполнения теста [20, 21], но не обнаружение несоответствия между реализацией системы и ее спецификациями. Для установления факта успешного прохождения теста, необходимо описать соответствующие проверки или формальную модель [3–5] (так называемая «проблема оракула» [9]).

Данная работа объединяет оба подхода: разработан интерактивный конструктор трасс, который, с одной стороны, упрощает поиск поведения соответствующего требованию путем задания контрольных точек, с другой – автоматически исследует альтернативы промежуточных участков и генерирует результирующий тестовый набор, удовлетворяющий структурному покрытию.

Разработан прототип для автоматизированного построения тестовых сценариев. Помимо соответствия требованиям высокого уровня, генерируемые тесты удовлетворяют критерию покрытия ветвей, а для участков поведения, требующих более тщательного тестирования, применен эффективный алгоритм [22] для покрытия потока данных [23]. Для повышения способности обнаруживать ошибки применен метод [24] конкретизации символьных трасс, осуществляющий вычисление и подстановку конкретных значений, лежащих на границах допустимого диапазона. Метод успешно применен [25] к анализу legacy-кода и генерации тестов [26] для Java программ.

Литература

1. Myers G.J. (2004). The Art Of Software Testing. New York. John Wiley & Sons, Inc. – 254 P.
2. Волков В., Колчин А., Летичевский А., Потиенко С. (2017). Обзор систематических методов генерации тестовых данных по исходному коду программных систем // Искусственный интеллект, 2, 71–84.
3. Fraser G., Wotawa F., Ammann P. (2009). Testing with model checkers: a survey // Software Testing, Verification and Reliability, 19, 215–261.
4. Utting M., Legeard B. (2007). Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann. – 456 P.
5. Petrenko A., Silva S., Maldonado J. (2012). Model-based testing of software and systems: recent advances and challenges // Software tools for technology transfer, 14(4), 383–386.
6. Rushby J. (2008). Automated test generation and verified software // Verified Software: Theories, Tools, Experiments, 161–172.
7. Gay G., Staats M., Whalen M., Heimdahl M. (2015). The risks of coverage-directed test case generation // IEEE Transactions on Software Engineering, 41, 803–819.
8. Heimdahl M., Whalen M., Rajan A., Staats M. (2008). On MC/DC and implementation structure:

An empirical study // In Proc. of Digital Avionics Systems Conf.

9. Barr E., Harman M., McMinn P., Shahbaz M., Yoo S. (2015). The oracle problem in software testing: a survey // IEEE Transactions on Software Engineering, 41, 507–525.
10. Athanasiou D., Nugroho A., Visser J. and Zaidman A. (2014). Test code quality and its relation to issue handling performance // IEEE Transactions on Softw. Eng, 40(11), 1100–1125.
11. Palomba F., Panichella A., and oth. (2016). Automatic Test Case Generation: What if Test Code Quality Matters? // In Proc. of Int. Symp. on Softw. Testing and Analysis, 130–141.
12. Beyer D., Dangl M. (2016). SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms // Verified Software. Theories, Tools, and Experiments, 181–198.
13. Baranov S., Kotlyarov V., Weigert T. (2012). Varifiable Coverage Criteria For Automated Testing. SDL2011: Integrating System and Software Modeling // LNCS, 7083, 79–89.
14. Колчин А.В., Котляров В.П., Дробинцев П.Д. (2012). Метод генерации тестовых сценариев в среде инсерционного моделирования // Управляющие системы и машины, 6, 43–48.
15. Колчин А.В. (2009). Разработка инструментальных средств для проверки формальных моделей асинхронных систем: Дис. ... канд. физ.-мат. наук. – Киев. –140 С.
16. Kolchin A.V. (2018). Interactive method for cumulative analysis of software formal models behavior // Proc. of the 11th Int. conf. of programming UkrPROG'2018, CEUR-WS, 2139, 115–123.
17. Колчин А.В. (2013). Метод редукции анализируемого пространства поведения при верификации формальных моделей распределенных программных систем // Искусственный интеллект, 4, 113–126.
18. Hessel A., Petterson P. (2007). A global algorithm for model-based test suite generation // Electr. Notes Theor. Comput. Sci, 190(2), 47–59.
19. Boonstoppel P., Cadar C. (2008). RWset: Attacking path explosion in constraint-based test generation. LNCS, 4963, 351–366.
20. Cseppento L., Micskei Z. (2015). Evaluating symbolic execution-based test tools // In IEEE Int. Conf. on Software Testing, Verification and Validation, 1–10.
21. Fraser G., Arcuri A. (2015). 1600 faults in 100 projects: automatically finding faults while achieving high coverage with Evosuite // Emperical software engineering, 20(3), 611–639.
22. Kolchin A. (2018). A novel algorithm for attacking path explosion in model-based test generation for data flow coverage // IEEE Int. Conf. on System Analysis & Intelligent Computing.
23. Su T., Wu K., Miao W., Pu G., and oth. (2017). A Survey on Data-Flow Testing // ACM Comput. Survey, 50(1), 35p.
24. Voinov N., Drobintsev P. and oth. (2015). Method of Symbolic Test Scenarios Automated Concretization // Proceedings of the Institute for system programming of the RAS, 27, 115–124.
25. Губа А., Колчин А., Потиенко С. (2016). Метод извлечения логики поведения из промышленного программного кода на языке Кобол // Проблемы программирования, 1–2, 17–25.
26. Колчин А.В., Потиенко С.В. (2016). Метод генерации тестовых данных по исходному коду Java программ // Искусственный интеллект, 3, 50–58.

References

1. Myers G.J. (2004). The Art Of Software Testing. New York. John Wiley & Sons, Inc. –254p.
2. Volkov V., Kolchin A., Letychevskiy A., Potiyenko S. Obzor sistematicheskih metodov generacii testovyh dannyh po ishodnomu kodu programmnyh sistem // Iskusstvennyj intellekt. –2017. –N2. –P. 71–84.
3. Fraser G., Wotawa F., Ammann P. (2009). Testing with model checkers: a survey // Software Testing, Verification and Reliability, 19, 215–261.
4. Utting M., Legeard B. (2007). Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann. – 456p.
5. Petrenko A., Silva S., Maldonado J. (2012). Model-based testing of software and systems: recent advances and challenges // Software tools for technology transfer, 14(4), 383–386.
6. Rushby J. (2008). Automated test generation and verified software // Verified Software: Theories, Tools, Experiments, 161–172.
7. Gay G., Staats M., Whalen M., Heimdahl M. (2015). The risks of coverage-directed test case generation // IEEE Transactions on Software Engineering, 41, 803–819.
8. Heimdahl M., Whalen M., Rajan A., Staats M. (2008). On MC/DC and implementation structure: An empirical study // In Proc. of Digital Avionics Systems Conf.
9. Barr E., Harman M., McMinn P., Shahbaz M., Yoo S. (2015). The oracle problem in software testing: a survey // IEEE Transactions on Software Engineering, 41, 507–525.
10. Athanasiou D., Nugroho A., Visser J. and Zaidman A. (2014). Test code quality and its relation to issue handling performance // IEEE Transactions on Softw. Eng, 40(11), 1100–1125.
11. Palomba F., Panichella A., and oth. (2016). Automatic Test Case Generation: What if Test Code Quality Matters? // In Proc. of Int. Symp. on Softw. Testing and Analysis, 130–141.
12. Beyer D., Dangl M. (2016). SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms // Verified Software. Theories, Tools, and Experiments, 181–198.
13. Baranov S., Kotlyarov V., Weigert T. (2012). Varifiable Coverage Criteria For Automated Testing. SDL2011: Integrating System and Software Modeling // LNCS, 7083, 79–89.
14. Kolchin A.V., Kotlyarov V.P., Drobintsev P.D. (2012). Metod generacii testovyh scenariev v srede

insercionnogo modelirovanija // Upravljajushhie sistemy i mashiny, 6, 43–48.
15. Kolchin A.V. (2009). Razrabotka instrumentalnykh sredstv dlya proverki formalnykh modeley asinkhronnykh sistem, Dis. … kand. fiz.-mat. nauk, Kiev. –140p.
16. Kolchin A.V. (2018). Interactive method for cumulative analysis of software formal models behavior // Proc. of the 11th Int. conf. of programming UkrPROG'2018, CEUR-WS, 2139, 115–123.
17. Kolchin A. (2013). Metod reduktsii analiziruemogo prostranstva povedeniya pri verifikatsii formalnyih modeley raspredelennyih programmnyih sistem // Iskusstvennyiy intellect, 4, 113–126.
18. Hessel A., Petterson P. (2007). A global algorithm for model-based test suite generation // Electr. Notes Theor. Comput. Sci, 190(2), 47–59.
19. Boonstoppel P., Cadar C. (2008). RWset: Attacking path explosion in constraint-based test generation. LNCS, 4963, 351–366.
20. Cseppento L., Micskei Z. (2015). Evaluating symbolic execution-based test tools // In IEEE Int. Conf. on Software Testing, Verification and Validation, 1–10.
21. Fraser G., Arcuri A. (2015). 1600 faults in 100 projects: automatically finding faults while achieving high coverage with Evosuite // Emperical software engineering, 20(3), 611–639.
22. Kolchin A. (2018). A novel algorithm for attacking path explosion in model-based test generation for data flow coverage // IEEE Int. Conf. on System Analysis & Intelligent Computing.
23. Su T., Wu K., Miao W., Pu G., and oth. (2017). A Survey on Data-Flow Testing // ACM Comput. Survey, 50(1), 35p.
24. Voinov N., Drobintsev P. and oth. (2015). Method of Symbolic Test Scenarios Automated Concretization // Proceedings of the Institute for system programming of the RAS, 27, 115–124.
25. Guba A., Kolchin A., Potiyenko S. (2016). Metod izvlechenija logiki povedenija iz promyshlennogo programmnogo koda na jazyke Cobol // Problemy programmirovanija, 1–2, 17–25.
26. Kolchin A.V., Potiyenko S.V. (2016). Metod generacii testovyh dannyh po ishodnomu kodu Java programm // Iskusstvennyj intellect, 3, 50–58.

RESUME

A. Kolchin, S. Potiyenko

Interactive method for automated test suit development for formal models of software systems

Software industry moves toward model-based development, and automated test generation from the model is often considered as a form of requirements-based testing. The majority of test generation approaches use some structural coverage criterion based on a behavioral model of the SUT to guide the selection of test cases. However, using the coverage provision as a target for automated test generation in many cases is a flawed strategy: coverage metrics are intended to measure the thoroughness of human-generated tests, and do not necessarily lead to good test sets when used in an inverted role as a specification for the tests required.

This paper proposes a novel interactive method for simplification and efficiency improvement of test scenario development to satisfy high-level requirements coverage. The method implements semiautomatic generation of tests on a basis of points-of-interest of desired behavior of a system under development. Unlike existing methods of models behavior analysis, which produce as a result only one witness or counter-example path per specified property, the proposed method provides analysis of behavior in a cumulative way: it generates aggregate information about all satisfiable paths. This distinctive feature plays a role of interactive path constructor, which prompts all satisfiable behavior alternatives, so user can find a desired path by iteratively specifying points-of-interest and observe updates of the prompting on-the-fly. The method is based on the original algorithm that, on the one hand, guarantees completeness of the search, and on the other it avoids exhaustive state-space exploration by applying a specialized decision procedure for early termination of path unfolding, which allows exploration of a state only if it might increase the requested coverage.

Надійшла до редакції 28.09.2018

